Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 61–78
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Context Dependent Procedures and Computed Types in √eriFun

Andreas Schlosser Christoph Walther Michael Gonder
Markus Aderhold

Fachgebiet Programmiermethodik
Technische Universität Darmstadt
Germany
http://www.informatik.tu-darmstadt.de/pm

Abstract

We present two enhancements of the functional language $\mathcal{L}$ which is used in the √eriFun system to write programs and formulate statements about them. *Context dependent procedures* allow to stipulate the context under which procedures are sensibly executed, thus avoiding runtime tests in program code as well as verification of absence of exceptions by proving stuck-freeness of procedure calls. *Computed types* lead to more compact code, increase the readability of programs, and make the well-known benefits of type systems available to non-freely generated data types as well. Since satisfaction of context requirements as well as type checking becomes undecidable, proof obligations are synthesized to be proved by the verifier at hand, thus supporting static code analysis. Information about the type hierarchy is utilized for increasing the performance and efficiency of the verifier.

Keywords: Context Dependency, Computed Types, Reasoning on Types, Subtyping.

1 Introduction

We develop the √eriFun system [1,14,15], an interactive system for the verification of statements about programs written in the functional first-order programming language $\mathcal{L}$ [12]. This language consists of definition principles for freely generated polymorphic data types, for procedures operating on these data types based on recursion, case analyses, let-expressions and functional composition, and for statements (called "lemmas" in $\mathcal{L}$) about the data types and the procedures. Procedures are evaluated in a call-by-value discipline. The data types *bool* with constructors *true* and *false*, and $\mathbb{N}$ for natural numbers with constructors 0 and $^+(\ldots)$ for the successor function are predefined in $\mathcal{L}$. Lemmas are defined by universal quantifications using case analyses and the truth values to represent connectives. Upon definition of a data type, each argument position of a constructor is provided with a selector function, e.g. $^-(\ldots)$ is the selector of constructor $^+(\ldots)$ thus representing the predecessor function, and *hd* and *tl* are the selectors of the *list*-constructor ::,

1571-0661
doi:10.1016/j.entcs.2006.10.038

```
structure bool <= true, false
structure ℕ <= 0, ⁺(⁻:ℕ)
structure list[@X] <= ε, [infix] :: (hd:@X, tl:list[@X])

function [outfix] | (k:list[@X]):ℕ <=
if ?ε(k) then 0 else ⁺(| tl(k) |) end

function [infix] !!(k:list[@X], n:ℕ):@X <=
if ?ε(k) then ⋆ else if ?0(n) then hd(k) else (tl(k) !! ⁻(n)) end end

function ordered(k:list[ℕ]):bool <=
if ?ε(k)
  then true
  else if ?ε(tl(k))
         then true
         else if hd(k) > hd(tl(k)) then false else ordered(tl(k)) end
end end

function find(key:ℕ, a:list[ℕ], i, j:ℕ):bool <=
if | a | > j
  then if j = i
    then key = (a !! i)
    else if j > i
      then let h := i + ⌈½(j − i)⌉ in
             let mid := (a !! h) in
               if mid > key
                 then find(key, a, i, ⁻(h))
                 else if key > mid then find(key, a, ⁺(h), j) else true end
               end end end
      else false end end
  else false
end

function binsearch(key:ℕ, a:list[ℕ]):bool <=
if ?ε(a) then false else find(key, a, 0, ⁻(| a |)) end

lemma binsearch is sound <= ∀a:list[ℕ], key:ℕ
  if{binsearch(key, a), key ∈ a, true}

lemma binsearch is complete <= ∀a:list[ℕ], key:ℕ
  if{ordered(a), if{key ∈ a, binsearch(key, a), true}, true}
```

Fig. 1. Searching a list by *binary search*

cf. Fig. 1. Type variables are preceded by "@" and expressions of form $?cons(x)$ are used as shorthand notation for $x = cons(sel_1(x), \ldots, sel_n(x))$, where *cons* is a constructor and sel_i are the selectors belonging to *cons*, hence e.g. $?::(x)$ holds iff list x is not empty.

The language also supports *incomplete definitions* of procedures [16] (also called *loose specifications* or *underspecifications* in the literature), using the symbol $\star$ to denote an indetermined result in cases which usually cause a runtime error or an exception. For example, procedure !! of Fig. 1 computes the n^{th} element of a list, where list elements are addressed from left to right starting with 0. Hence the result of $(k \,!!\, n)$ is indetermined (denoted by $\star$ in the body of !!) if $|k| \leq n$. Incomplete definitions are also used to define *abstract mappings* like the arity function $\| \ldots \|$ of Fig. 3.

Figure 1 gives an example of an $\mathcal{L}$-program for searching in an ordered list by the *binary search method* as well as the lemmas stating soundness and completeness of the search procedure, cf. [13].

2 Context Dependent Procedures

2.1 Context Clauses

We call procedures which may only be executed in a certain context *context dependent*. Context dependency is an "old theme" in computer science. Almost any programming language provides some mechanism for stipulating context dependency, assembly languages, early dialects of Lisp etc. being the rare exceptions. Nowadays, programming languages provide elaborate type systems which allow to check context properties upon compile time. Our focus when talking about context dependency are requirements which—differently from type checking—usually are undecidable so that theorem proving is needed to check satisfaction of the context constraints.

Generally, the context requirement for a procedure is a predicate over the formal parameters of the procedure. If this requirement is not satisfied in a calling context, the result of the procedure call—if any—may be arbitrary. Consequently, satisfaction of the context requirement in the calling context is a necessary prerequisite that the procedure behaves in the intended way.

A *context dependent procedure* f of an $\mathcal{L}$-program P is defined by expressions of the form

$$\texttt{function}\ f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau <= \texttt{assume}\ c_f; body_f \quad . \tag{1}$$

The *context clause* $c_f \in \mathcal{T}(\Sigma(P), \{x_1, \ldots, x_n\})_{bool}$ given by the `assume` declaration is represented by a boolean term built with the function symbols (except f) defined by the data type and procedure definitions of program P (given by the signature $\Sigma(P)$ of P) and the formal parameters x_i of procedure f. A context clause c_f defines a precondition which needs to hold when executing procedure calls of f. Since the `assume` declaration is optional, *true* is used by default for c_f if no context clause is explicitly provided in the procedure body.

The semantics of a context dependent procedure is simply defined as the semantics of its *relativized* version `function` $f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau$ <= $body_f^{ctx}$, where $body_f^{ctx}$:= *if* c_f *then* $body_f$ *else* $\star$ *end* denotes the *relativized body* of f. Since all selectors are incompletely defined, selectors are assigned a context clause, too: For a constructor *cons* with the corresponding selectors $sel_1,\ldots,sel_n$, the context clause c_{sel_i} of a selector call $sel_i(x)$ is defined as $?cons(x)$ stating that a selector must only be applied to the constructor it belongs to.

For instance, procedure !! of Fig. 1 can be reformulated using a context clause as displayed in Fig. 2. This context clause demands that procedure calls $(k\,!!\,n)$ only appear in contexts which guarantee that n is a legal address of list k. As $|k| > n$ entails $k \neq \varepsilon$, the test for $?\varepsilon(k)$ (and the indetermined result $\star$ in turn) is omitted in the body of the context dependent version of procedure !!, hence the absence of exceptions when calling !! now is guaranteed *statically* by the context clause. Consequently, execution of procedure !! is *more efficient*, as the $\min(|k|, n)$ tests for $?\varepsilon(k)$ are saved in the context dependent version.

The definition of procedure *find* is refined as well in Fig. 2. The context clause demands $|a| > j \geq i$ for the upper bound j and the lower bound i of the search interval in list a. Also here, a more efficient procedure is obtained as tests performed dynamically upon each (recursive) call of procedure *find* are replaced by a static test: The tests for $|a| > j$ and $j > i$ are saved, both of which cause costs proportional to $\lceil \log_2(j - i + 1) \rceil$ in the original version of find.

2.2 Context Hypotheses

To guarantee that procedures (or selectors) f are called in a valid context within a term t only, a so-called *context requirement* is generated for t expressing that the context under which any f is called in t entails the context clause c_f of f (with formal parameters in c_f replaced by the actual parameters of the call).

Definition 2.1 [Context Requirements] For a term t, the set $Octx(t) \subseteq Occ(t)$ is the set of all *context sensitive occurrences* in t, i.e. occurrences $\pi \in Occ(t)$ such that $t_{|\pi} = g(t_1,\ldots,t_n)$ and g is a selector or a procedure function symbol. [1]

The *context requirement* $CR(t)$ of term t is given as $AND(\{CR(t,\pi) | \pi \in Octx(t)\})$, where $CR(t,\pi) = if\{COND(t,\pi), \theta(c_g), true\}$ if $t_{|\pi} = g(t_1,\ldots,t_n)$, and θ replaces the formal parameters x_i of g by the actual parameters t_i. [2]

For a *procedure* f as in (1), the *context hypothesis* of f is defined as `lemma` $f\$ctx$ <= $\forall x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ $CR(body_f^{ctx})$, and the *context hypothesis* of a lemma

$$\texttt{lemma}\ lem <= \forall x_1{:}\tau_1,\ldots,x_k{:}\tau_k\ body_{lem} \tag{2}$$

is defined as `lemma` $lem\$ctx$ <= $\forall x_1{:}\tau_1,\ldots,x_k{:}\tau_k$ $CR(body_{lem})$.

[1] As usual, $Occ(t)$ is the set of all *occurrences* of t, $t_{|\pi}$ denotes the subterm of t at occurrence π, and term $t[\pi \leftarrow r]$ is obtained from t by replacing $t_{|\pi}$ by r.

[2] $AND(\{b_1,\ldots,b_n\})$—sometimes also written as $AND(b_1,\ldots,b_n)$—abbreviates a boolean term representing the conjunction of the boolean terms b_i. $COND(t,\pi)$ is the conjunction of all conditions in t leading to subterm $t_{|\pi}$.

✓eriFun demands that the context hypothesis of a procedure f be verified *before* verification of lemmas and (other) procedures "calling" f starts. Also the context hypothesis of a lemma must be verified *before* verification of the lemma.

For example, using the context clauses of *hd*, *tl*, $^-(\ldots)$, and !!, context hypothesis !!$\$ctx$ is generated for the context dependent procedure !! of Fig. 2, and (using the context clauses of $^-(\ldots)$ and *find*) context hypothesis *binsearch*$\$ctx$ is generated for the context dependent version of procedure *binsearch*. The context hypothesis for lemma *binsearch is complete* of Fig. 1 is displayed in Fig. 2 as well.

2.3 Determination Hypotheses

When calling an *incompletely* defined procedure, so-called *stuck computations* may result. For each user-defined procedure `function` $f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau$ `<=` ..., ✓eriFun synthesizes a so-called *domain procedure* `function` $\nabla f(x_1{:}\tau_1,\ldots,x_n{:}\tau_n)$:*bool* `<=` ... . Domain procedures ∇f are always completely defined—i.e. stuck computations never occur when calling ∇f—and provide an equivalent requirement for the absence of stuck computations when calling the "mother" procedure f, i.e. computation of $\nabla f(q_1,\ldots,q_n)$ yields *true* iff computation of $f(q_1,\ldots,q_n)$ does not get stuck, see [16] for details.

Figure 2 displays domain procedure ∇!! synthesized by ✓eriFun for procedure !!.[3] Hence absence of stuck computations when calling $(k\,!!\,n)$ is guaranteed iff computation of $\nabla!!(k,n)$ yields *true*. But as domain procedure ∇!! just provides a recursive definition for deciding $|k|>n$, computation of $(k\,!!\,n)$ does not get stuck iff $|k|>n$ is satisfied upon a procedure call $(k\,!!\,n)$.

Using domain procedures, absence of stuck computations can be determined *statically*. To this effect, a so-called *determination hypothesis* `lemma` $f\$det$ `<=` $\forall x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ *if*$\{c_f,\nabla f(x_1,\ldots,x_n),\mathit{true}\}$ is generated for each context dependent procedure as given in (1). Determination hypotheses simply express that a procedure's *context clause* is *sufficient* for the *absence of stuck computations*. Hence the truth of a context requirement of a call of procedure f entails the absence of stuck computations for this call, provided f's determination hypothesis has been verified.[4] As an example, Fig. 2 displays the determination hypothesis !!$\$det$ generated for procedure !!.

3 Computed Types

3.1 Non-freely Generated Data Types

Data types in $\mathcal{L}$ are *freely generated*, which means that the semantics of *monomorphic* types τ of an $\mathcal{L}$-program P—either defined directly, such as *bool* and $\mathbb{N}$, or otherwise obtained as a monomorphic instance of a non-monomorphic data type, such as *list*[$\mathbb{N}$], *list*[*list*[$\mathbb{N}$]], etc.—can be defined as a free term algebra with carriers

[3] When synthesizing ∇f for a context dependent procedure f, $body_f^{ctx}$ is to be used.

[4] Verification of determination hypotheses cannot generally be *demanded*, as calls of abstract mappings like procedure $\|\ldots\|$ in Fig. 3 are inherently indetermined.

```
function [infix] !!(k:list[@X], n:ℕ):@X <=
assume | k | > n; if ?0(n) then hd(k) else (tl(k) !! ⁻(n)) end

lemma !!$ctx <= ∀k:list[ℕ], n:ℕ
  if{| k | > n,
    if{?0(n), ?::(k), if{?::(k), if{?⁺(n), | tl(k) | > ⁻(n), false}, false}},
    true}

function ∇!!(k:list[@X], n:ℕ):bool <=
if | k | > n
  then if ?0(n)
       then ?::(k)
       else if ?::(k) then ∇!!(tl(k), ⁻(n)) else false end
     end
  else false
end

lemma !!$det <= ∀k:list[ℕ], n:ℕ if{| k | > n, ∇!!(k, n), true}

function find(key:ℕ, a:list[ℕ], i, j:ℕ):bool <=
assume if{i > j, false, | a | > j};
if j = i
  then key = (a !! i)
  else let h := i + ⌈½(j − i)⌉ in let mid := (a !! h) in
         if mid > key
           then find(key, a, i, ⁻(h))
           else if key > mid then find(key, a, h, j) else true end
       end end end
end

function binsearch(key:ℕ, a:list[ℕ]):bool <=
assume ordered(a); if ?ε(a) then false else find(key, a, 0, ⁻(| a |)) end

lemma binsearch$ctx <= ∀a:list[ℕ]
  if{ordered(a),
    if{?ε(a), true, if{?⁺(| a |), if{0 > ⁻(| a |), false, | a | > ⁻(| a |)}, false}},
    true}

lemma binsearch is complete$ctx <= ∀a:list[ℕ], key:ℕ
  if{ordered(a), if{key ∈ a, ordered(a), true}, true}
```

Fig. 2. Context dependent procedures

$\mathcal{T}(\Sigma(P)^c)_\tau$ over a signature $\Sigma(P)^c$ of constructor function symbols. However, often one is concerned with proper subsets of $\mathcal{T}(\Sigma(P)^c)_\tau$ only. For example, the set of prime numbers $\mathbb{P} \subset \mathbb{N}$ is used when working in number theory, and lists of "type" *list*[$\mathbb{P}$] are returned by a procedure doing prime factorization. Also *normal forms* are used quite frequently, for example for efficient search: Procedure *binsearch* of Fig. 1, e.g., operates on ordered lists, where *ordered* is a proper subset of the freely generated data type *list*[$\mathbb{N}$].

All these examples have in common that certain structures s one is concerned with—prime numbers, normal forms, etc.—are not freely generated but rather are proper subsets of some freely generated data types τ. Consequently, "recognizer procedures" `function` $s(x{:}\tau){:}bool$ <= ... must be used to decide whether an item of type τ is a member of s. Subsequently, we shall use such procedures like freely generated data types, and we call them *computed types*, as algorithmic definitions are provided for them.

Definition 3.1 [Computed Types] Let $\mathcal{P}$ be the set of all *polymorphic types* of an $\mathcal{L}$-program P, let $\mathcal{W} \subseteq \mathcal{P}$ be the set of *type variables*, and let $\mathcal{M} \subseteq \mathcal{P}$ be the set of all *monomorphic types*.[5] We extend the type system of P by a set $\mathsf{M} := \bigcup_{i\in\mathbb{N}} \mathsf{M}_i$ where $\mathsf{M}_0 := \mathcal{M}$ and $\mathsf{M}_{i+1} := \mathsf{M}_i \cup \{\gamma \in \Sigma(P) \mid \gamma : \delta \to bool \text{ for some } \delta \in \mathsf{M}_i\} \cup \{\zeta[\gamma_1, \ldots, \gamma_n] \mid \zeta \text{ is an } n\text{-ary type constructor and } \gamma_1, \ldots, \gamma_n \in \mathsf{M}_i\}$. P is the set of all polymorphic types where instantiation of type variables with computed types from $\mathsf{M} \setminus \mathcal{M}$ is allowed (but not necessary). The *base type* $\mathcal{P}(\gamma) \in \mathcal{P}$ of a type $\gamma \in \mathsf{P}$ is defined as γ if $\gamma \in \mathcal{M} \cup \mathcal{W}$, $\mathcal{P}(\gamma) := \mathcal{P}(\delta)$ if $\gamma \in \mathsf{M}_{i+1}$ and $\gamma : \delta \to bool$ for some $\delta \in \mathsf{M}_i$, and $\mathcal{P}(\zeta[\gamma_1, \ldots, \gamma_n]) := \zeta[\mathcal{P}(\gamma_1), \ldots, \mathcal{P}(\gamma_n)]$.

The *$\mathcal{P}$-subtype relation* $\leqslant_\mathcal{P}$ is defined as the reflexive and transitive closure of the *direct $\mathcal{P}$-subtype relation* $\lessdot_\mathcal{P} \subseteq \mathcal{P} \times \mathcal{P}$, which is defined as the smallest relation satisfying

(i) $\xi(\tau) \lessdot_\mathcal{P} \tau$, if $\xi = \{@v_i/\zeta[@w_1, \ldots, @w_n]\}$ for some $@v_i \in \mathcal{W}(\tau)$, $@w_1, \ldots, @w_n \notin \mathcal{W}(\tau)$ where $@w_1, \ldots, @w_n$ are pairwise different and ζ is an n-ary type constructor,[6] and

(ii) $\xi(\tau) \lessdot_\mathcal{P} \tau$, if $\xi = \{@v_i/@v, @v_j/@v\}$ where $@v_i, @v_j \in \mathcal{W}(\tau)$, $@v \notin \mathcal{W}(\tau)$.

The *M-subtype relation* $\leqslant_\mathsf{M}$ is defined as the reflexive and transitive closure of the *direct M-subtype relation* $\lessdot_\mathsf{M} \subseteq \mathsf{P} \times \mathsf{P}$, which is defined as the smallest relation satisfying

(iii) $\gamma \lessdot_\mathsf{M} \delta$, if $\gamma \in \mathsf{M}_{i+1}$ and $\gamma : \delta \to bool$ for some $\delta \in \mathsf{M}_i$ and

(iv) $\zeta[\gamma_1, \ldots, \gamma_n] \lessdot_\mathsf{M} \zeta[\delta_1, \ldots, \delta_n]$, if $\gamma_j \lessdot_\mathsf{M} \delta_j$ for some $j \in \{1, \ldots, n\}$, $\gamma_k = \delta_k$ for each $k \in \{1, \ldots, n\} \setminus \{j\}$, and ζ is an n-ary type constructor.

The *subtype relation* $\leqslant_\mathsf{P} := \leqslant_\mathsf{M} \circ \leqslant_\mathcal{P} \subseteq \mathsf{P} \times \mathsf{P}$ is defined as the composition of the M-subtype relation $\leqslant_\mathsf{M}$ and the $\mathcal{P}$-subtype relation $\leqslant_\mathcal{P}$.

The subtype relation $\leqslant_\mathsf{P}$ defines a join-semilattice $(\mathsf{P}, \leqslant_\mathsf{P})$ on the set of poly-

[5] $\mathcal{L}$ uses *parametric* polymorphism [3]. Since $\mathbb{N}$ is predefined in $\mathcal{L}$, $\mathcal{M}$ is not empty.

[6] $\mathcal{W}(\tau)$ denotes the set of all type variables used in type τ.

morphic and computed types (modulo α-conversion, i.e. type variable renaming). Hence, for $\tau_1, \tau_2 \in \mathsf{P}$ the most special common super type is uniquely determined as the least upper bound $\tau_1 \sqcup \tau_2$ of types τ_1 and τ_2, i.e. $\tau_1, \tau_2 \leqslant_{\mathsf{P}} \tau_1 \sqcup \tau_2 \leqslant_{\mathsf{P}} \tau$ for each τ with $\tau_1, \tau_2 \leqslant_{\mathsf{P}} \tau$.

Figure 3 shows some computed types in the domain of the (freely generated) data type *pterm*.[7] Procedures $wft \in \mathsf{M}_1$, $wfgt \in \mathsf{M}_2$ and $wfgtu \in \mathsf{M}_3$ (meaning "well-formed term", "well-formed ground term" and "well-formed ground unit application") define computed types, where $\mathcal{P}(wfgtu) = \mathcal{P}(wfgt) = \mathcal{P}(wft) = pterm$ and $wfgtu \lessdot_{\mathsf{M}} wfgt \lessdot_{\mathsf{M}} wft \lessdot_{\mathsf{M}} pterm \lessdot_{\mathcal{P}} @a$. Since $pair[vsym, wft] \in \mathsf{M}_2$, $list[pair[vsym, wft]] \in \mathsf{M}_3$, and $wfsubst \in \mathsf{M}_4$, where $pair[vsym, wft] \lessdot_{\mathsf{M}} pair[vsym, pterm]$, the type hierarchy $wfsubst \lessdot_{\mathsf{M}} list[pair[vsym, wft]] \lessdot_{\mathsf{M}} list[pair\ [vsym, pterm]] \lessdot^{+}_{\mathcal{P}} list[pair[@x, @y]] \lessdot_{\mathcal{P}} list[@z] \lessdot_{\mathcal{P}} @q$ can be established for *wfsubst*, and $\mathcal{P}(wfsubst) = \mathcal{P}(list[pair[vsym, wft]]) = list[pair[vsym, pterm]]$.

3.2 Type Checking with Computed Types

Since type checking becomes undecidable with the involvement of computed types, it has to be supported by theorem proving. To be able to check whether some term is of type $\zeta[\gamma_1, \ldots, \gamma_n] \in \mathsf{P} \setminus \mathcal{P}$, a so called *sort instance procedure* $\zeta\$\gamma_1\$\cdots\$\gamma_n : \mathcal{P}(\zeta[\gamma_1, \ldots, \gamma_n]) \to bool$ is synthesized which returns *true* iff its argument is of type $\zeta[\gamma_1, \ldots, \gamma_n]$. Sort instance procedures are defined recursively using the recursion structure of data type ζ. For example, procedure $list\$pair\$vsym\$wft$ is synthesized to check for all elements of its input list recursively if their second component is a well-formed term, cf. Fig. 3.

To check whether some term t has type $\gamma \in \mathsf{P}$, $t : \gamma$ for short, a boolean term $TC(t, \gamma)$ is created by defining

- $TC(t, \gamma) := true$, if $\gamma \in \mathcal{P}$ and $t : \gamma$, [8]
- $TC(t, \gamma) := if\{TC(t, \delta), \gamma(t), false\}$, if $\gamma \in \Sigma(P)$ and $\gamma \lessdot_{\mathsf{M}} \delta$, and
- $TC(t, \zeta[\gamma_1, \ldots, \gamma_n]) := if\{TC(t, \zeta[\delta_1, \ldots, \delta_n]), \zeta\$\gamma_1\$\cdots\$\gamma_n(t), false\}$, if ζ is an n-ary type constructor and $\zeta[\gamma_1, \ldots, \gamma_n] \lessdot_{\mathsf{M}} \zeta[\delta_1, \ldots, \delta_n]$. [9]

For instance, for verifying $q : wfgtu$ wrt. the procedures of Fig. 3, $if\{wft(q), if\{wfgt(q), wfgtu(q), false\}, false\}$ is computed for $TC(q, wfgtu)$ (after *if*-terms have been normalized) and $if\{list\$pair\$vsym\$wft(\sigma), wfsubst(\sigma), false\}$ is the normalized result for $TC(\sigma, wfsubst)$.

Now to check for type correctness when calling a procedure as given in (1)—where, however, $\tau_i \in \mathsf{P}$ now is allowed—the boolean expression $AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n))$ is added conjunctively to the context clause c_f of procedure f. This guarantees implicitly that all actual parameters have the correct type in each procedure call whose context requirement is verified. Hence, computed types extend

[7] Data type *pterm* defines *packed terms* unifying terms and termlists into one structure. E.g., term $\mathrm{F}(\mathrm{G}(\mathrm{A},\mathrm{B}),\mathrm{C},\mathrm{D})$ is represented by $apply(\mathrm{F}, pack(apply(\mathrm{G}, pack(\mathrm{A},\mathrm{B})), pack(\mathrm{C},\mathrm{D})))$.

[8] As $\gamma \in \mathcal{P}$ in this case, side condition "$t : \gamma$" is decided by "conventional" type checking.

[9] Type $\zeta[\delta_1, \ldots, \delta_n]$ is not uniquely determined by $\lessdot_{\mathsf{M}}$, but since the order in which the type hierarchy is traversed is irrelevant, an arbitrary direct super type of $\zeta[\gamma_1, \ldots, \gamma_n]$ is chosen.

```
structure pair[@X, @Y] <= [infix] • ([postfix] 1:@X, [postfix] 2:@Y)
structure vsym <= variable(v.index :ℕ)
structure csym <= constant(c.index :ℕ)
structure fsym <= function(f.index :ℕ)
structure pterm <= var(vbl:vsym), const(cst:csym),
                   apply(func:fsym, argument:pterm), pack(left:pterm, right:pterm)

function [outfix] ‖ (f:fsym):ℕ <= ⋆

function #(t:pterm):ℕ <= if ?pack(t) then #(left(t)) + #(right(t)) else 1 end

function wft(t:pterm):bool <=
case t of
  apply : if ‖ func(t) ‖ = #(argument(t)) then wft(argument(t)) else false end
  pack : if ?pack(left(t))
           then false
           else if wft(left(t)) then wft(right(t)) else false end
         end
  other : true
end

function wfgt(t:wft):bool <=
case t of
  apply : wfgt(argument(t))
  pack : if wfgt(left(t)) then wfgt(right(t)) else false end
  other : ?const(t)
end

function wfgtu(t:wfgt):bool <=
if ?apply(x) then ‖ argument(t) ‖ = 1 else false end

function wfsubst(σ:list[pair[vsym, wft]]):bool <=
if ?ε(σ)
  then true
  else if ?pack((hd(σ))2) then false else wfsubst(tl(σ)) end
end

function list$pair$vsym$wft(x:list[pair[vsym, pterm]]):bool <=
if ?ε(x)
  then true
  else if wft((hd(x))2) then list$pair$vsym$wft(tl(x)) else false end
end
```

Fig. 3. Computed types in the *pterm* domain

$$\texttt{function}\ binsearch(key{:}\mathbb{N}, a{:}ordered){:}bool\ <=$$
$$if\ \ ?\varepsilon(a)\ then\ false\ else\ find(key, a, 0, {}^{-}(|\,a\,|))\ end$$

$$\texttt{lemma}\ binsearch\ is\ complete <= \forall a{:}ordered, key{:}\mathbb{N}$$
$$if\{key \in a, binsearch(key, a), true\}$$

$$\texttt{lemma}\ binsearch\ is\ complete\$ctx <= \forall a{:}ordered, key{:}\mathbb{N}$$
$$if\{key \in a, ordered(a), true\}$$

Fig. 4. Binary search with computed types

the parametric polymorphism of $\mathcal{L}$ by *inclusion* polymorphism [3]. The semantics of a procedure with computed types is defined as the semantics of its *relativized* version

$$\texttt{function}\ f(x_1{:}\mathcal{P}(\tau_1), \ldots, x_n{:}\mathcal{P}(\tau_n)){:}\tau\ <=$$
$$if\ \ AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n), c_f)\ then\ body_f\ else\ \star\ end \quad .$$

Similarly, lemmas as given in (2) can be defined using computed types by allowing $\tau_i \in \mathsf{P}$. Also here, the semantics of a lemma using computed types is defined as the semantics of its *relativized* version

$$\texttt{lemma}\ lem <= \forall x_1{:}\mathcal{P}(\tau_1), \ldots, x_n{:}\mathcal{P}(\tau_n)$$
$$if\{AND(TC(x_1, \tau_1), \ldots, TC(x_n, \tau_n)), body_{lem}, true\} \quad .$$

$$\texttt{function}\ apply.to.var(\sigma{:}wfsubst, v{:}vsym){:}wft\ <=$$
$$if\ \ ?\varepsilon(\sigma)$$
$$\quad then\ var(v)$$
$$\quad else\ if\ \ v = (hd(\sigma))_1\ then\ (hd(\sigma))_2\ else\ apply.to.var(tl(\sigma), v)\ end$$
$$end$$

$$\texttt{function}\ apply.to.term(\sigma{:}wfsubst, t{:}wft){:}wft\ <=$$
$$case\ t\ of$$
$$\quad var : apply.to.var(\sigma, vsym(t))$$
$$\quad const : t$$
$$\quad apply : apply(func(t), apply.to.term(\sigma, argument(t)))$$
$$\quad pack : pack(apply.to.term(\sigma, left(t)), apply.to.term(\sigma, right(t)))$$
$$end$$

Fig. 5. Computed types as result types

Figure 4 displays a refinement of procedure *binsearch* using procedure *ordered* of Fig. 1 as a computed type. Also lemma *binsearch is complete* uses the computed type *ordered*, thus allowing a more compact representation of the lemma. By the refinement of procedure *binsearch*, *ordered*(a) arises as an additional (trivial) proof obligation in the context hypothesis *binsearch is complete\$ctx*.

Computed types are also allowed as *result types* of procedures, i.e. $\tau \in \mathsf{P}$ instead of $\tau \in \mathcal{P}$ may be allowed as well in procedures as given in (1). In such a case, a so-called *signature hypothesis* is synthesized as an $\mathcal{L}$-lemma

$$\texttt{lemma}\ f\$sig <= \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ TC(f(x_1, \ldots, x_n), \tau)$$

in ✓eriFun.[10] The semantics of a procedure with a computed type τ as result type is simply defined as the semantics of a procedure which has its base type $\mathcal{P}(\tau)$ as result type.

Procedures *apply.to.var* and *apply.to.term* of Fig. 5 are examples for procedures with computed result types. The signature hypotheses for these procedures simply express that *well-formed* terms are obtained if *well-formed* substitutions are applied to *variables* or *well-formed* terms respectively, where the signature hypothesis generated for *apply.to.var* is required to verify the signature hypothesis for *apply.to.term*.

4 Reasoning with Computed Types

So far, the use of computed types only leads to more compact and readable definitions of procedures and lemmas in $\mathcal{L}$. This means that some syntactic sugar has been spread on our programming language for easing its use. Verification comes into play only to support type checking when computed types are involved, viz. for proving context, determination and signature hypotheses.

However, the main value of computed types comes with utilization of type information upon reasoning, as this may support verification considerably. The situation is quite comparable with the benefit of conventional type systems which do not only support writing of more compact and more readable code, but allow to detect program faults at compile time rather than at runtime only.

4.1 The HPL-*Calculus*

Statements about procedures of an $\mathcal{L}$-program are formulated in ✓eriFun as $\mathcal{L}$-lemmas, i.e. expressions of the form given in (2). The proof of a lemma usually requires induction, the base and step formulas of which are represented by *sequents* of the form $h_1, \ldots, h_n; \forall ih_1, \ldots, \forall ih_m \Vdash goal$ where $\{h_1, \ldots, h_n\}$ denotes the set of *hypotheses* defining the base or step case respectively. The set of *induction hypotheses* of a step case is given by $\{\forall ih_1, \ldots, \forall ih_m\}$, where the non-induction variables are universally quantified, and *goal*, called the *goalterm* of the sequent, represents the induction conclusion. The induction hypotheses and the goalterm are boolean terms, and the hypotheses are literals.[11]

[10] Usually, signature hypotheses have a straightforward proof by induction according to the recursion structure of procedure f. Hence termination of f is verified before verification of the signature hypothesis begins.

[11] An atom a is an *if*-free boolean term, and a literal is an atom or a negated atom written as $\neg a$ or as $t \neq r$ for negated equations $t = r$. $\overline{l}$ stands for the complement of a literal l and $\overline{C} := \{\overline{l} \mid l \in C\}$ for a clause C.

The set of sequents defines the language of the *HPL-calculus* (abbreviating *Hypotheses*, *Programs* and *Lemmas*), which is the calculus in which the lemmas are proved. The application of a proof rule of this calculus to a sequent yields a finite set of sequents, which are obtained by altering the set of hypotheses, the set of induction hypotheses or the goalterm of the sequent to which the proof rule has been applied. Each proof rule is *sound* in the sense that the truth of all resulting sequents entails the truth of the sequent to which the proof rule has been applied. A proof in the *HPL*-calculus is represented by a *prooftree*, the nodes of which are given by sequents. The root node of a prooftree for a lemma *lem* is given by the *initial sequent* $\Vdash body_{lem}$, and the successor nodes are given by the sequents resulting from a proof rule application to the father node sequent. A proof of lemma *lem* is obtained if a *closed* prooftree can be built for *lem*, i.e. a prooftree where each leaf is a sequent of the form $\ldots \Vdash true$.

The *HPL*-calculus provides a set of 15 proof rules to create prooftrees [8,15]. For example, *Induction* creates the base and step sequents from an initial sequent wrt. some induction axiom, *Use Lemma* applies a lemma to a sequent, *Case Analysis* creates successor sequents by a case split, *Unfold Procedure* "opens up" a procedure call in a goalterm, etc. Some of these proof rules require a *substitution* or a *term* as input, and in order to ensure well-formedness of the resulting sequent, syntactical requirements for these inputs have to be checked. Hence, with the use of computed types, these requirements need to be updated.

To supply the *HPL*-calculus with type information about the used variables, sequents are extended by an additional set of *type hypotheses*. Thus, a sequent with type hypotheses has the form

$$th_1, \ldots, th_l;\ h_1, \ldots, h_n;\ \forall ih_1, \ldots, \forall ih_m \Vdash goal \tag{3}$$

where the type hypotheses are literals of form $\gamma(x_i)$ with $\gamma \in \mathsf{P}$, where $\gamma(x_i) := true$ is omitted if $\gamma \in \mathcal{P}$.[12] The initial sequent of a prooftree for a lemma *lem* as given in (2) is $\tau_1(x_1), \ldots, \tau_k(x_k);\ ;\ \Vdash body_{lem}$.

When applying the *Induction* rule for using induction upon variables $x_1 : \gamma_1, \ldots, x_k : \gamma_k$ of (possibly *computed*) types γ_i, an additional sequent

$$th_1, \ldots, th_l;\ h_1, \ldots, h_n; \Vdash AND\left(\bigcup_{j=1}^{m}\bigcup_{i=1}^{k}\{TC(t_{j,i}, \gamma_i)\}\right) \tag{4}$$

is created for each step sequent of the form (3), demanding *type correctness* for each substitution $\{x_1/t_{j,1}, \ldots, x_k/t_{j,k}\}$ used to build the induction hypothesis ih_j (under the hypotheses h_i defining the step case). For instance, when proving $\forall x{:}ordered$

[12] If γ has the form $\zeta[\gamma_1, \ldots, \gamma_n]$ the corresponding type hypothesis is actually built with the sort instance procedure of γ.

goal by structural *list*-induction upon x, the sequents

$$ordered(x);\ \ ?\varepsilon(x);\Vdash goal \tag{5}$$

$$ordered(x);\ \ ?::(x);\ \ goal[x/tl(x)]\Vdash goal \tag{6}$$

$$ordered(x);\ \ ?::(x);\Vdash ordered(tl(x)) \tag{7}$$

are generated as successor sequents of the initial sequent, where sequents (5) and (6) are the usual base and step sequents coming with the *list*-induction, and (7) is the *additional sequent* coming with (4) for guaranteeing type correctness of the substitution used to form the induction hypothesis.

But if $\forall x{:}list.ev\ goal$ is to be proved instead for a computed type $list.ev$, where $list.ev(x)$ iff the length of list x is even, structural *list*-induction becomes unsound, as $tl(x) : list.ev$ is false if $x : list.ev$ and $? :: (x)$ holds. As the additional step sequent $list.ev(x);\ ?::(x);\Vdash list.ev(tl(x))$ cannot be proved, the prooftree obtained by *list*-induction upon $x : list.ev$ cannot be closed, thus avoiding an unsound induction proof. If, however, $\forall x{:}list[@X]\ goal$ is considered, the additional step sequent $; ? :: (x);\Vdash if\{true, true, true\}$ is obtained which trivially simplifies to *true*, thus imposing no restriction upon the *list*-induction.

Also additional successor sequents have to be generated for other *HPL*-proof rules which expect terms or substitutions as input in order to guarantee *type correctness* as well as *context correctness* (thus entailing *well-typedness* in particular, cf. Sect. 3.2) of the rule input. For example, when employing *Use Lemma* to apply an instance $\sigma(body_{lem})$ of an $\mathcal{L}$-lemma as given in (2) to a sequent's goalterm, the successor sequent $\ldots \Vdash goal[\pi \leftarrow if\{\sigma(body_{lem}), goal_{|\pi}, true\}]$ is obtained for a sequent as given in (3). [13] Substitution $\sigma = \{x_1/t_1, \ldots, x_k/t_k\}$ (which must be provided when calling *Use Lemma*) replaces the universally quantified variables $x_i : \tau_i$ of lemma *lem* by terms t_i which use variables of the sequent only, and ✓eriFun's parser is used to check whether each t_i is a well-formed term of type τ_i. Now when allowing *computed* types so that $\tau_i \in \mathsf{P}$ may hold as well, type checking has to be supported by theorem proving. To this effect, the system generates the proof obligation

$$wtc_\sigma = if\{COND(goal, \pi), AND\left(\bigcup_{i=1}^{k}\{CR(t_i), TC(t_i, \tau_i)\}\right), true\}$$

expressing that—under the context $COND(goal, \pi)$ of the lemma application—each t_i is a *context correct* term of *type* τ_i. The system then demands verification of proof obligation wtc_σ simply by providing father sequent (3) with an additional successor sequent $\ldots \Vdash wtc_\sigma$. As the truth of *all* successor sequents is demanded for the father sequent to hold, proof obligation wtc_σ must be verified in order to obtain a closed proof tree.

Finally, a new *HPL*-rule *Relativize* has to be provided in addition since some

[13] We write $\ldots \Vdash goal$ if a successor sequent inherits all hypotheses, all type and all induction hypotheses of the father sequent. $\pi \in Occ(goal)$ is a parameter of the *Use Lemma* rule, where π is restricted to occurrences allowing terms of type *bool* only.

verification problems require making type information explicit: If $goal_{|\rho} : \gamma$ holds for the subterm of *goal* at a user-provided occurrence ρ, *Relativize* creates the successor sequent $\ldots \Vdash goal[\pi \leftarrow if\{TC(goal_{|\rho}, \gamma), goal_{|\pi}, true\}]$ for a sequent of the form (3), where π selects the smallest superterm of $goal_{|\rho}$ with type *bool*.

For instance, when proving the context requirement for the recursive call of procedure *apply.to.var* of Fig. 5, the sequent

$$wfsubst(\sigma); \ldots \Vdash if\{?\varepsilon(\sigma), true, wfsubst(tl(\sigma))\} \tag{8}$$

arises. In order to prove (8), it is necessary to unfold the procedure call $wfsubst(\sigma)$ which is *implicitly* provided by the type hypothesis. To this effect, *Relativize* applied to the type hypothesis of (8) creates a new successor sequent

$$\ldots \Vdash if\{wfsubst(\sigma), if\{?\varepsilon(\sigma), true, wfsubst(tl(\sigma))\}, true\} \quad .$$

Now the procedure call occurs explicitly in the goal term, thus being available for unfolding now.

4.2 The Evaluation Calculus

Goalterms of a sequent (3) are simplified by so-called *computed HPL*-proof rules. For instance, *Simplification* rewrites the sequent's goalterm using the definitions of the data types and procedures, the hypotheses and the induction hypotheses of the sequent and the lemmas already verified. These rewrites are performed by *symbolic evaluation* which is defined by another calculus, called the *evaluation calculus*, cf. [8, 17]. The language of this calculus is given by the set $\mathcal{T}(\Sigma(P), \mathcal{V})$ of first-order terms, where $\Sigma(P)$ stands for the signature of the function symbols defined by an $\mathcal{L}$-program P, and $\mathcal{V}$ is a set of typed variables used in the sequents. The inference rules of the evaluation calculus, called *evaluation rules*, are of the form "$\frac{term}{term'}$, if COND", where COND stands for a side condition which must be satisfied for applying the evaluation rule. We write $term \vdash_{H,A} term'$ if $term'$ originates from *term* by an evaluation rule using a set H of literals containing at least the sequent's hypotheses h_i and type hypotheses th_k, and clauses from a finite set $A \subset \mathcal{CL}(\Sigma(P), \mathcal{V} \cup \mathcal{U})$ which represents the sequent's induction hypotheses ih_j as well as the verified lemmas of P and are built with skolemized typed variables from $\mathcal{V}$ and universally quantified variables from a set $\mathcal{U}$ of typed variables. Thus, the set A contains type information as well, in particular the proven signature and context hypotheses.

Symbolic evaluations are computed in ✓`eriFun` by the *Symbolic Evaluator*, i.e. an automated theorem prover which considers the evaluation rules in a fixed order. Similarly to the *HPL*-calculus, some of the evaluation rules have to be modified. However, differently to the *HPL*-calculus, these updates are not only required for guaranteeing context and type correctness, but to utilize the knowledge about computed types and the subtype relation $\leqslant_{\mathsf{P}}$ for improving performance of the *Symbolic Evaluator*.

The rule *Affirmative hypothesis* of Fig. 6 is extended such that it simplifies terms by considering $\leqslant_{\mathsf{P}}$. For example, $wft(t) \vdash_{H,A} true$ is obtained in one evaluation step

Affirmative hypothesis

$$\frac{a}{true}\ \text{, if } a \in H \text{ or } a = \delta(d), \gamma \leqslant_{\mathsf{P}} \delta \text{ and } \gamma(d) \in H$$

Evaluate then part

$$\frac{if\{a,b,c\}}{if\{a,b',c\}}\ \text{, if } \begin{cases} b \vdash_{H',A} b' \\ \text{where } H' = H \cup \{a\} \setminus \{\gamma(d) \in H | \delta \leqslant_{\mathsf{P}} \gamma\} \\ \text{if } a = \delta(d) \text{ for some } \delta \in \mathsf{P} \setminus \mathcal{P} \\ \text{and } H' = H \cup \{a\} \text{ else} \end{cases}$$

Affirmative assumption

$$\frac{a}{true}\ \text{, if } \begin{cases} a = \sigma_\xi(lit) \text{ and } lit' \vdash^{+}_{H \cup \{\neg a\},A} false \ \text{ for some } \sigma_\xi, \\ \text{some } D \in A, \text{ some } lit \in D, \\ \text{some } \theta_{\xi'} \text{ with } \mathcal{U}(\theta_{\xi'}(\sigma_\xi(D))) = \emptyset \\ \text{and all } lit' \in \theta_{\xi'}(\sigma_\xi(D \setminus \{lit\})) \end{cases}$$

Assumption replacement

$$\frac{t}{if\{TC(\sigma_\xi(r),\tau), \sigma_\xi(r), t\}}\ \text{, if } \begin{cases} t = \sigma_\xi(l) \text{ for some } \sigma_\xi, \text{ some } D \in A, \\ \text{some } l \texttt{=>} r \in D, \text{ some } \theta_{\xi'} \\ \text{with } \theta_{\xi'}(\sigma_\xi(D \setminus \{l \texttt{=>} r\})) \subseteq \overline{H}, \text{ and } t : \tau \end{cases}$$

Fig. 6. Evaluation rules using computed types

for the computed types of Fig. 3 if $wfgtu(t) \in H$, instead of exploring the direct subtype relation $\leqslant_{\mathsf{M}}$ step by step as it would be required otherwise. In addition, the rule *Evaluate then part* is modified such that it extends the set H of hypotheses under consideration of the subtype relation. If a new type hypothesis $\gamma(t)$ is to be added to H, unnecessary type hypotheses $\delta(t)$ with $\gamma \leqslant_{\mathsf{P}} \delta$ are removed from H. This keeps the set H as small as possible.

The evaluation rules for using clauses from the clause set A are modified as well to incorporate the subtype relation $\leqslant_{\mathsf{P}}$. Central to this modification is the use of $\leqslant_{\mathsf{P}}$ for matching: A pattern type $\tau_1 \in \mathsf{P}$ matches a target type $\tau_2 \in \mathsf{P}$ *modulo computed types*, iff a type substitution ξ with $\mathcal{P}(\xi(\tau_1)) = \mathcal{P}(\tau_2)$ and $\tau_2 \leqslant_{\mathsf{P}} \xi(\tau_1)$ exists. Substitution ξ is the matcher of τ_1 and τ_2, iff $\xi(\tau_1) \sqcup \tau_2 \leqslant_{\mathsf{P}} \xi'(\tau_1) \sqcup \tau_2$ for every type substitution ξ'. A pattern term t_1 matches a target term t_2 *modulo computed types* iff there exists a term substitution σ and a type substitution ξ (modulo computed types) such that $\sigma_\xi(t_1) = t_2$.

Now evaluation rule *Affirmative assumption* for using verified clauses from A is modified by incorporation of *matching modulo computed types*. Hence the universally quantified variables of clauses from A can be matched with terms t_i having

$\leqslant_{\mathsf{P}}$-smaller types in the goalterm. The types of terms t_i are determined using the type hypotheses in H and the signature information in A. These reasoning steps are obviously sound, as each property which is true in the domain of type δ holds in the domain of computed type $\gamma \leqslant_{\mathsf{P}} \delta$ as well. The benefit of matching modulo types stems from the fact that reasoning about subtypes is shifted into the matching algorithm instead of performing the required reasoning steps explicitly by several proof steps using the evaluation rules to refute subtype predicates. The dual evaluation rule *Negative assumption* which replaces a redex by *false* if σ_ξ is a matcher for *lit* and $\neg a$ uses matching modulo computed types as well. Similarly, evaluation rule *Assumption replacement* for equality reasoning using oriented equations $l \mathsf{=>} r$ is updated to use matching modulo computed types, thus yielding the same benefits as for the *Affirmative/Negative assumption* rules. By adding a local type condition it is ensured that the replaced term has the correct type in the context of the rule application.

5 Related Work

Context dependency has been investigated for verifying termination of loops in imperative programs [2,4]. Loops are translated here into tail-recursive procedures for which context requirements—called *termination predicates*—are *synthesized* which are sufficient for the procedures' termination. A verifier then is used to prove that the program context (given by the properties of the program variables used in the loop) entails the synthesized termination predicate. Termination predicates can be expressed by the context clauses of our proposal, which, however, must be supplied explicitly to the program code, see e.g. requirement $j \geq i$ for procedure *find* of Fig. 2.

The functional programming language *Miranda* supports definition of non-free data types as subtypes of free data types by stipulation of so-called *laws* [9], i.e. data types are enhanced by rewrite rules transforming values into normal forms, e.g. lists into ordered lists. This approach guarantees that values are always rewritten into normal forms, but there are no restrictions imposed on functions operating on these types, since the normal form has not to be preserved by the functions, but is restored by the *laws* automatically. Hence, type correctness is not enforced by type checking but ensured by rewriting. This approach is limited to subtypes which represent normal forms. For instance, subtypes like *wft* of Fig. 3 cannot be handled with laws.

ACL2 supports subtyping by so-called *guards* [6]. Guards are predicates which are used to check the arguments of functions for type correctness, corresponding to computed types in our setting. Guards are only available for the definition of procedures. Subtypes can neither be used in lemma definitions nor as result types of procedures. The main benefit of guards in ACL2 is to verify absence of exceptions upon the execution of a Common Lisp program, thus corresponding to the proofs of *determination hypotheses* in our proposal.

PVS represents types by sets, and *computed* types can be defined as subsets

via predicates [7], thus providing a framework most comparable to our approach. Differently to our proposal, PVS supports *polymorphic* computed types as well *dependent types*, which merge context dependent procedures and computed types into a single concept. However, reasoning about computed types is always performed explicitly by verifying PVS proof obligations, whereas it is partially incorporated into the reasoning machinery in our proposal (thus utilizing well-known benefits from classical theorem proving, see [10, 11] and see [18] for an exhaustive account on subsequent developments).

6 Conclusion

We presented two enhancements—implemented in an experimental system [5]—of the functional language $\mathcal{L}$ which is used in the ✓`eriFun` system to write programs and formulate statements about them. *Context dependent procedures* allow to specify the context under which procedures are sensibly executed, thus avoiding run-time tests in program code, which would be required otherwise. The tests whether a procedure is called in a program environment which guarantees the procedure's context demand can be performed statically. Proof obligations are generated to be proved by the verification system at hand. Context dependent procedures are also used to verify absence of exceptions statically by proving stuck-freeness of procedure calls. *Computed types* lead to more compact code and increase readability of programs, and the well-known benefits of type systems in programming languages become available for non-freely generated data types as well. Information about the type hierarchy is utilized for increasing performance and efficiency of the verifier. As type-checking becomes undecidable, proof obligations are synthesized. To show type correctness and well-typedness of expressions, these proof obligations are proved by the verifier.

Context dependent procedures and computed types do not subsume each other: Using context dependent procedures only, restrictions on the return type of procedures cannot be formulated, which is accomplished by computed types. Using computed types only prevents the specification of execution contexts for procedures which cannot be expressed by unary predicates.

Presently, we are investigating various upgrades of our proposal: We intend to allow *composition* of computed types by the set-theoretic operators $\cap$ and $\cup$ to write, for example, `function` *prime factors*$(n : \mathbb{N}) : \mathit{list}[\mathbb{P}] \cap \mathit{ordered} <= \ldots$. Also *dependent computed types* as provided by PVS are under investigation to define a computed type in dependence of procedure parameters or lemma variables respectively. For instance, using such a feature one may write for procedure *find* of Fig. 2 `function` $\mathit{find}(\mathit{key}{:}\mathbb{N}, a{:}\mathit{list}[\mathbb{N}] \cap |\_| > j, i{:}\mathbb{N}, j{:}\_ \geq i){:}\mathit{bool} <= \ldots$ where a is assigned the dependent type $\mathit{list}[\mathbb{N}] \cap |\_| > j$ and j is assigned the dependent type $\_ \geq i$, altogether expressing $|a| > j \geq i$. Whether such expressions increase readability of programs is a matter of taste (and of use, of course). As the requirements for the parameters a and j in the example can be expressed by context dependent procedures as well, cf. Fig. 2, the key question is here whether dependent computed

types contribute to *automated* reasoning in the same way as computed types do. Finally, it has to be investigated whether *polymorphic* computed types as provided by PVS are a useful enhancement.

References

[1] http://www.verifun.org.

[2] Brauburger, J. and J. Giesl, *Approximating the Domains of Functional and Imperative Programs*, Science of Computer Programming **35** (1999), pp. 113–136.

[3] Cardelli, L. and P. Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys **17** (1985), pp. 471–522.

[4] Giesl, J., C. Walther and J. Brauburger, *Termination Analysis for Functional Programs*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction - A Basis for Applications, vol. 3* (1998), pp. 135–164.

[5] Gonder, M., "Entwurf und Implementierung kontextabhängiger Prozeduren und Untertypen in √veriFun," Diploma thesis, Programmiermethodik, TU Darmstadt (2006).

[6] Kaufmann, M., P. Manolios and J. S. Moore, "Computer-Aided Reasoning: An Approach," Kluwer Academic Publishers, 2000.

[7] Rushby, J., S. Owre and N. Shankar, *Subtypes for Specifications: Predicate Subtyping in PVS*, IEEE Transactions on Software Engineering **24** (1998), pp. 709–720.

[8] Schweitzer, S., "Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme," Doctoral dissertation, Programmiermethodik, TU Darmstadt (to appear).

[9] Thompson, S., *Laws in Miranda*, in: *Proceedings of the 1986 ACM conference on LISP and functional programming (LFP)* (1986), pp. 1–12.

[10] Walther, C., "A Many-Sorted Calculus Based on Resolution and Paramodulation," Research Notes in Artificial Intelligence, Pitman and Morgan Kaufmann, London and Los Altos, 1987.

[11] Walther, C., *Many-Sorted Unification*, Journal of ACM **35** (1988), pp. 1–17.

[12] Walther, C., M. Aderhold and A. Schlosser, *The ℒ 1.0 Primer*, Technical Report VFR 06/01, Programmiermethodik, TU Darmstadt (2006).

[13] Walther, C. and S. Schweitzer, *A Verification of Binary Search*, Technical Report VFR 02/02, Programmiermethodik, TU Darmstadt (2002).

[14] Walther, C. and S. Schweitzer, *About √veriFun*, in: F. Baader, editor, *19th International Conference on Automated Deduction (CADE)*, LNAI **2741** (2003), pp. 322–327.

[15] Walther, C. and S. Schweitzer, *Verification in the Classroom*, Journal of Automated Reasoning **32** (2004), pp. 35–73.

[16] Walther, C. and S. Schweitzer, *Reasoning about Incompletely Defined Programs*, in: G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, LNAI **3835** (2005), pp. 427–442.

[17] Walther, C. and S. Schweitzer, *A Pragmatic Approach to Equality Reasoning*, Technical Report VFR 06/02, Programmiermethodik, TU Darmstadt (2006).

[18] Weidenbach, C., *Sorted Unification and Tree Automata*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction - A Basis for Applications, vol. 1* (1998), pp. 291–320.